DOI:10.19651/j.cnki.emt.2210472

面向仪用总线的实时 Sunday 数据帧提取算法设计*

范正吉 党立志 逯玉玉 洪应平 张会新 储成群
(中北大学仪器与电子学院 太原 030000)

摘 要: 在各类仪用总线网络通信过程中,系统往往需要在应用层上对接收的高速数据流进行实时处理。而如何对连续的数据流进行数据帧数据提取是讨论的主要问题。对此,分析了常见仪用总线的协议处理方法,并设计了一套帧提取算法,算法包含帧提取状态机、改进的 Sunday 帧头匹配算法以及帧内子域查找算法。然后本文采用直接发送、经由 TCP 网络发送两种环境对算法进行测试,实验证明本算法性能优于 Netty 框架下的帧长度域解码。最后,本文为了实际测试和应用算法,使用该算法对 64 通道,100 kS/s 模拟量采集卡进行数据帧的实时提取和存储,并对采集的模拟量进行波形显示。本算法可用于仪用总线应用层的数据分隔、帧头识别、帧数据提取工作。

关键词: Sunday 算法;数据帧提取;模式匹配;Netty 框架
中图分类号: TH701 **文献标识码:** A **国家标准学科分类代码:** 510.4

Design of real-time Sunday data frame extraction algorithm for instrument bus

Fan Zhengji Dang Lizhi Ti Yuyu Hong Yingping Zhang Huixin Chu Chengqun
(College of Instrument and Electronics, North University of China, Taiyuan 030000, China)

Abstract: In the communication process of various types of instrumentation bus networks, the system often needs to process the received high-speed data stream in real time on the application layer. And how to extract data frame data from continuous data stream is the main problem discussed. In this regard, analyzes the protocol processing methods of common instrument buses, and designs a set of frame extraction algorithms, including frame extraction state machine, improved Sunday frame header matching algorithm and intra-frame subdomain search algorithm. Then this paper tests the algorithm in two environments: direct sending and sending via TCP network. Experiments show that the performance of the algorithm is better than the frame length domain decoding under the Netty framework. Finally, in order to actually test and apply the algorithm, this paper uses the algorithm to extract and store the data frame in real time for the 64-channel, 100 kS/s analog acquisition card, and display the waveform of the acquired analog quantity. This algorithm can be used for data separation, frame header identification and frame data extraction at the application layer of the instrument bus.

Keywords: Sunday algorithm; data frame extraction; pattern matching; Netty framework

0 引 言

仪器仪表总线技术目前逐渐向高数据带宽,各仪器设备可重复利用,具备精确时间同步能力的方向发展。而在各类仪器总线网络通信过程中,在物理层上可能采用不同的传输介质,例如光纤、双绞线、同轴电缆等,但是在更高层次的处理方法中一般都需要具备单帧数据的发送、接收、分隔、路由、校验等功能[1]。而在应用各类总线网络的过程中,系统往往需要面向字节数据流对总线网络数据进行实时处理。例如在 RS422、RS232、RS485 等异步串行总线中,其上层计算机处理系统所获取的数据往往是单个字节或多个字节所组成的数组,而对于以太网传输协议中的传输控制协议(transmission control protocol, TCP)、用户数据报协议(user datagram protocol, UDP),具有操作系统的计算机一般通过 Socket 对象相关获取有效数据报文,其数据形式亦为若干字节。

为此,在实时数据流的基础上还需要额外的应用层协议进行更为具体和实际的数据传输(如 Modbus 协议、

收稿日期:2022-06-25
* 基金项目:山西省高等学校科技创新项目(2019L0539)资助

PROFINET 协议等)[2]。面向不同的应用场合,其应用层的帧协议是多样的。而在总线数据接收的过程中,如何对一串连续的数据流进行分割,并从中提取有效帧数据是本文讨论的主要问题。

针对连续的数据流进行数据帧数据提取的问题,学者和工程师根据不同协议和应用场景设计了处理算法。上海交通大学的王扬德等人提出了面向比特流的隐式马尔科夫模型通信协议识别方案,能够从比特级实现对已经协议以及未知协议的分析和识别[3]。赵恒永等[4]基于确定有限自动机(deterministic finite automation,DFA),设计了针对可编程逻辑控制器(programmable logic controller,PLC)、分散式控制系统(distributed control system,DCS)等工业自动化系统的自适应协议识别算法,通过用户配置文件对数据进行不同操作。Trustin 等在其所编写的网络应用程序框架——Netty 中构建了基于文件传输协议(file transfer protocol,FTP)、简单邮件传输协议(simple mail transfer protocol,SMTP)、超文本传输协议(hyper text transfer protocol,HTTP)等不同网络通信协议的数据帧解码器[5-7]。在以上研究中,在算法设计上针对特定的应用场景或特定的总线类型,且除却 Netty 框架外,面向高速实时的数据流往往存在处理耗时长、容易出现丢帧或帧内数据缺失等问题。对此,本文提出了基于 Sunday 的实时数据帧提取算法以用于通用的字节数据流处理。

本文首先对常见的总线的数据帧结构及协议处理方法进行归纳,得出总线协议中帧处理的特性,并介绍了应用层通常的帧分析方法,并在其基础上设计了一套通用的帧提取算法。然后本文与直接的状态机跳转方法、Netty 类似算法进行了基于 UDP 的数据帧处理实验结果比较,实验证明在帧长度较长时,本算法性能优于直接的状态机跳转方法,并整体优于 Netty 网络架构下帧长度域解码器。最后,本文为了实际测试和应用算法,使用 16 位,64 通道,100 kS/s 同步并行模拟量采集卡,在 PC 端使用该算法进行数据帧实时提取和存储,并对其采集的模拟量进行波形显示。

1　常见仪用总线的帧处理方法

本文分别对常见的仪用总线的分析,如通用串行总线(universal serial bus,USB)、LAN 面向仪器的扩展总线(LAN-based extensions for instrumentation,LXI)、以太网、RS422、RS485 等,可以归纳得出各类总线在帧处理具有如下的特性:

1) 在总线通信中有一定的确定数据帧起始的方案,一般有特定的帧起始标识符、特定的同步电平等,即便没有帧起始的标志,也需要一定的帧分隔时间用以确定帧的起始,例如 Modbus 协议。

2) 若多个设备处于同一总线网络结构中,则帧结构一般需要地址域信息。

3) 若总线中存在不同类型的帧,则帧结构一般需要帧类型信息。

4) 总线可采用循环冗余校验(cyclic redundancy check,CRC)、和校验、奇偶位校验等方式对整帧数据进行校验。

5) 帧本身长度一般由 3 种方式确定:协议采用固定长度、帧头帧尾界定、数据中长度域信息界定。

即总线通信协议中,帧收发的方面一般具有 5 点内容:帧起始、帧地址、帧类型、数据校验、帧长度,其中帧起始、帧长度是必要的。

而对于系统而言,不论是异步串行类总线,还是基于以太网的总线,系统获取总线数据往往本质上是从固定存储空间或先进先出队列(first input first output queue,FIFO)缓冲中获取的[8-9],系统单次接收到的是单个字节或若干字节所组成的数组。此时若在该数据流中提取出应用层的数据帧,则系统同样需要确定以上处理帧的 5 点内容。对此,本文给出了各类总线提取帧数据的一般方案,如下:

首先,针对帧起始的处理,若应用层协议起始数据帧的方案为帧间隔时间对帧进行分隔,则在程序设计上可以较为简单地使用定时器对数据接收时间进行判定。而若不是间隔时间对帧进行分隔,或者在数据接收过程中,由于数据进入 FIFO 缓冲后丢失了间隔时间的信息,则在应用层协议中数据帧往往需要指定帧头(帧首标识符)来确定帧的起始位置。

其次,对帧长度的界定,若系统采用固定长度的帧,则等待接收长度一致的数据即可。而若采用帧头帧尾形式或采用长度域信息界定时,则一般需要状态机进行接收帧的状态判定,并采用一定的缓存空间存储当前的数据帧。

最后,对于帧地址、帧类型、数据校验的处理,系统在数据接收完成后在帧的对应位置进行相应的数据变换、判断即可。

在接收到字节数据流后,一般情况下系统会遍历字节数组,依次接收单个字节,并按照类似如图 1 所示的状态机进行帧的提取,状态机处理的帧格式如图 2 所示。

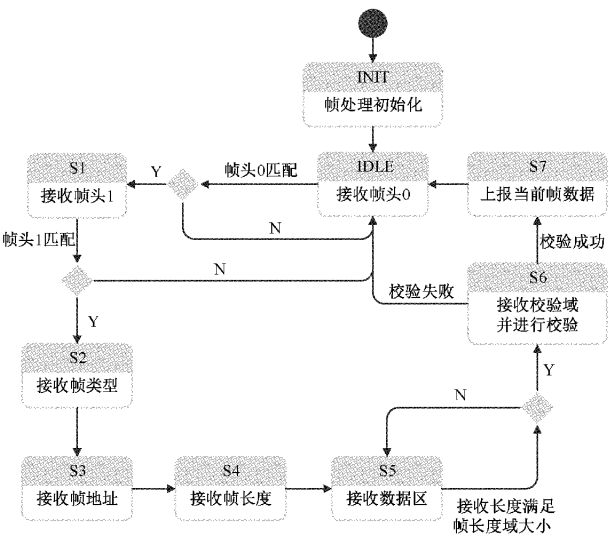

图 1　一般情况下接收处理数据帧的状态机

图 2 在图 1 中的状态机所处理的帧格式

处理数据帧的状态机的形式是多样的:帧中各个子域(帧头、帧类型、帧长度、数据区、校验域等)可能有多个字节,因此需要为多字节的子域定义多个状态;系统只需要个别设备的数据帧,因而需要依据地址域对数据帧进行筛选;各个子域的前后顺序可能不同;在帧的接收时可能需要进行超时处理,因而需要对当前状态的超时时间进行判断等。面对不同的帧格式和通信协议,其状态机形式各异,对于程序编程人员来说是繁复的。同时,若帧的长度较长,例如接收图像帧时,遍历字节数组并为每个字节判断状态是低效的。为此,本文提出了一套较为通用的帧提取算法来解决这个问题。

2 帧提取状态机设计

在上节叙述中,帧头和帧长度的界定是用于分割多个数据帧并处理的关键所在,而本文要处理的难点在于解决数据帧被被划分到两个甚至多个字节数组的情况,这点在 TCP 传输过程中是较为常见的,即 TCP 的粘包和拆包问题。因此,对于本算法而言,需要完成以下 5 步处理:

步骤 1)将帧头匹配问题转换为模式匹配问题,通过一定的模式匹配算法找出当前字节数组中的帧头。找帧头的过程中也需要考虑帧头被划分到两个甚至多个字节数组的情况。本文采用了一种改进的 Sunday 算法来帧头查找。这里该问题的解决详见下一节。

步骤 2)在找到帧头后找寻帧长度信息,对于固定长度的帧,则将剩余数据接收即可;对于具有帧头帧尾格式的帧,只需要在缓冲一帧数据的同时再次依照模式匹配算法找寻帧尾;而对于拥有长度域信息的帧,则需要缓冲帧并根据长度域的帧内偏移位置获取帧长度信息。

步骤 3)在获取到帧长度信息后进行帧内任意区域接收并进行帧缓存,这里考虑到在实际使用过程中上层应用可能只需要帧内的部分数据进行处理,此时系统就无需对帧内后续数据进行缓存和处理。

步骤 4)将用户所需要的帧区域交由上层或进入帧的 FIFO 缓冲队列。

步骤 5)根据帧长度信息计算帧结束偏移量,并在该帧之后进行下一帧的提取处理。

由此,本文设计了如图 3 状态机进行帧提取。

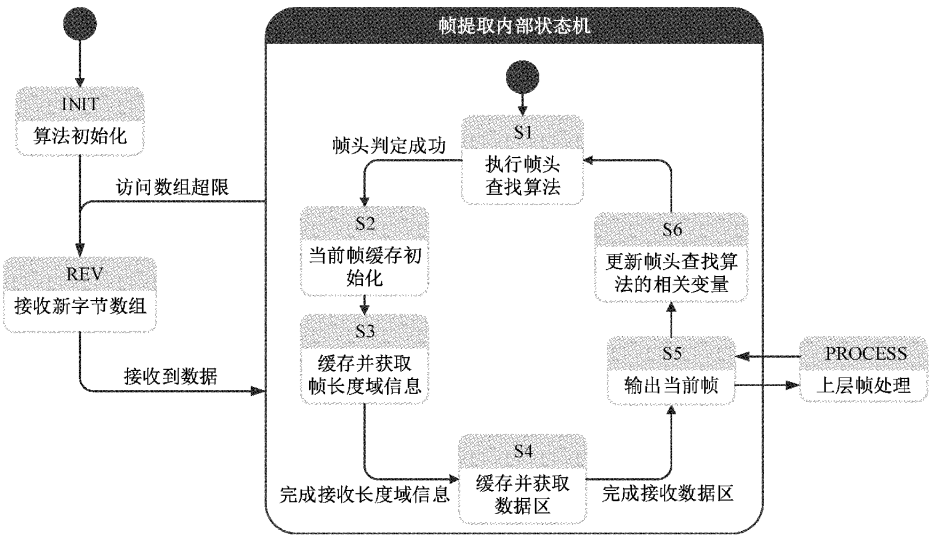

图 3 帧提取状态机

在该状态机中,接收到任何大小的字节数据都会经由帧提取内部状态机进行处理。内部状态机会依次运行 S1 到 S6 状态,并在 S1、S3、S4 状态访问到超过数组范围的内容时会跳出当前状态机然后等待下次数据接收。而下次接收到新数据时则再次会进入内部状态机并继续以当前状态处理,以确保处理数据的连续性。

在得到完整一帧数据后会在 S5 状态输出当前帧或帧的数据区,上层若采用 C/C++/C# 等语言编程时,可定义一个帧内容的结构体,并直接通过将结构体起始指针指向当前输出内容获取帧内各部分数据。

以下本文便详细说明帧头查找、帧缓存及帧内信息获取的算法。

3　改进的 Sunday 帧头匹配算法

帧头匹配问题可以理解为模式匹配问题，模式匹配算法的定义为：在一个长度为 n 字符串 S 中，寻找匹配的模式串 P。如果只需要匹配帧头，该算法则为单模式匹配问题；若需要匹配帧头、帧尾，则该算法为多模式匹配问题，即找出这些模式串在主串中的所有位置。

而针对模式匹配，目前已有不少匹配速度较快的算法，例如快速模式匹配算法(knuth-morris-pratt algorithm，KMP)、Sunday 算法、哈希检索算法（robin-karp algorithm，RK)、Boyer-Moore 算法(BM)等[10-11]。但是，对于系统处理过程而言，数据流是不断更新的，如果一个模式匹配算法在执行过程需要回退当前查找指针，则有可能进入上次处理的字节数组中，这会需要额外的数据缓存机制并导致程序过于复杂，所以 KMP 算法、BM 算法在这里并不适用。而对 RK 算法而言，通过对主串中 $n-m+1$ 个子串分别求 hash 值，然后与模式串 P 的哈希值进行比对来得出匹配的子串位置。而计算 Hash 值需要为每个子串计算，导致额外的运算量。对此，本文采用高效且无需回退跳转的 Sunday 算法作为帧头匹配的主体算法。

3.1　Sunday 算法简介

Daniel 提出了 Sunday 算法，该算法是从前到后进行匹配的，其主要思想如下[12]：

若模式串长度 m，当前配对位置为 I_{now}，若在当前位置匹配失败时，则查看参与匹配的最末字节的下一字节(该位置为 $I_{now}+m$)是否在模式串 P 中出现过，若未出现过，则下次配对位置为 $I_{next}=I_{now}+m+1$，即移动位数 $Shift=m+1$；若出现过，则该移动位数与匹配字节在模式串出现的位置有关，设其模式串中在最右出现的位置为 i，则移动位数 $Shift=m-i$。

这里以模式串‘uiop’为例，说明 Sunday 算法匹配流程。首先在当前比较位置匹配失败时，如图 4 所示，Sunday 算法会查看位置 4 的字符‘l’是否在模式串 P 内，因不包含‘l’则下次比较位置跳转至图 5 位置 $I_{next}=5$。

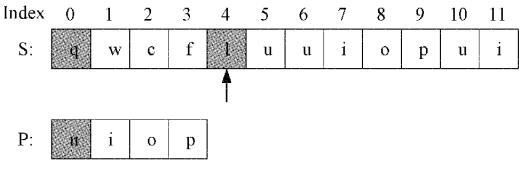

图 4　匹配失败时移动情况 1

而当当前查看位置 9 的字符‘p’在字符串 P 内时，则计算下次移动位数 $Shift=m-i=1$，即跳转至图 6 位置 $I_{next}=6$。

为了让 Sunday 算法应用于帧提取，则需要在匹配成功后需要继续进行下帧的帧头查找，设本帧长度为 Len，则下次比较位置为 $I_{next}=I_{now}+Len$。

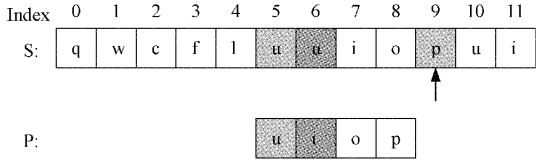

图 5　匹配失败时移动情况 2

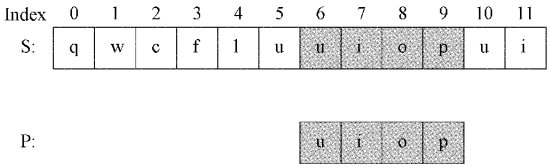

图 6　Sunday 算法匹配成功

3.2　适用于数据流的改进 Sunday 算法

当每次接收字节数组大小较小时，帧头被划分到多个字节数组是较为常见的，因此本算法所要解决的主要问题是在这种情况依然能进行匹配。对此，本文提出了适用于数据流的改进 Sunday 算法，该算法的输入为不断更新的字节数组，匹配到帧头时跳转帧内部处理状态机。

首先需要对算法内部变量初始化，主要需要初始化跳转表 ***Shift***，该表表示当前查看的字节到下次移动位数的映射关系，由于字节值的范围为 0～255，故初始化时进行遍历运算。其伪代码如下：

初始化改进 Sunday 算法

```
1：Input：待匹配帧头 HeadWord，帧头长度为 m
2：I_head ← 0
3：I_next ← 0
4：I_now ← 0
5：Offset ← 0
6：新建大小 255 数组 Shift
7：新建大小 m 的数组 Buf
8：for i ← 0 to 255 do
9：        Shift[i] ← m + 1
10：end for
11：for i ← 0 to m do
12：        Shift[HeadWord[i]] ← m − i
13：end for
```

完成初始化后，即可以数据流为输入运行该算法，伪代码如下：

改进 Sunday 算法(运行于帧提状态机中)

```
1：Input：接收的字节数组 DataIn，数据长度 L
2：while I_next < L do
3：      if I_next ≥ 0 then
4：        Judge ← DataIn[I_next]
5：      else
```

```
6:            Judge ← Buf[I_now − I_head]
7:     end if
8:     if Judge = HeadWord[Offset] then
9:      Buf[Offset] ← HeadWord[Offset]
10:  if Offset = 0 then
11:            I_head ← I_now
12:  end if
13:   Offset ← Offset + 1
14:  if Offset = m then
15:          Offset ← 0
16:              I_now ← I_now + m
17:                帧头匹配,帧提取状态机到 S2 状
18:                break
19:            end if
20:          I_next ← I_now + Offset
21:     else
22:            NL ← I_now + m
23:            Offset ← 0
24:            if NL < L then
25:                I_now ← I_now + Shift[DataIn[NL]]
26:        else
27:          I_now ← I_now + 1
28:            end if
29:          I_next ← I_now
30:     end if
31: end while
32: if I_next ≥ L then
33:    I_now ← I_now − L
34:      I_next ← I_next − L
35:      I_head ← I_head − L
36: end if
```

其中，I_{now}、I_{next} 及 I_{head} 分别为当前查找的指针在当前数据中的位置、下次查找的位置及当前查找到的帧头的位置。若 I_{next} 在查找帧头过程中若超过了当前数组大小，则会跳出当前循环并等待下次数据接收，同时 I_{now}、I_{next} 及 I_{head} 一同会自行减去当前数组大小。这里，三者的值可以是负值，代表之前接收的数据的偏移。该算法主要对两种特殊情况进行处理：

情况 1：考虑到可能出现帧头有一部分在当前字节数组的末尾的情况，在当前数据末尾执行进行依次比对（代码 27 ～ 29 行）。如图 7 所示，假设待匹配帧头为 ***HeadWord*** = {0xAA,0xAF,0x10,0x20}。

情况 2：假帧头情况。即在当前字节数组的末尾匹配到部分帧头，而在下次接收的数据开头发现是该帧头数据并不匹配。这种情况较少出现，一般存在于数据传输过程中，出现了数据的丢失，而丢失的刚好包含了帧头的一部分；或者在帧的数据区内包含部分帧头数据时。

本文对这种情况的处理方法是建立存放帧头数据的小缓存区，若匹配失败，则从缓存区的第二个字节继续进行匹配（代码 3～7 行）。如图 8 所示，假帧头为 ***HeadWord*** = {0xAA,0xAF,0x6F}。

4 帧内子域查找算法

在帧提取状态机中，S2、S3、S4 状态需要在找到帧头后缓存并提取帧内的长度域和数据域，对此，本文设计了一套较为简单的算法实现该功能。

首先，在 S2 状态对帧内子域查找算法进行初始化。设立当前字节数组的帧起始位置变量 I_{seg}，用于表示在当前字节数组内，接收到帧头后第一个字节的位置，即 I_{seg} = 帧头位置＋帧头长度。该值同样可为负值，表示代表之前接收的数据的偏移。同时设立当前帧缓存的长度 L_{seg}，以及帧缓存 ***SegBuf***。

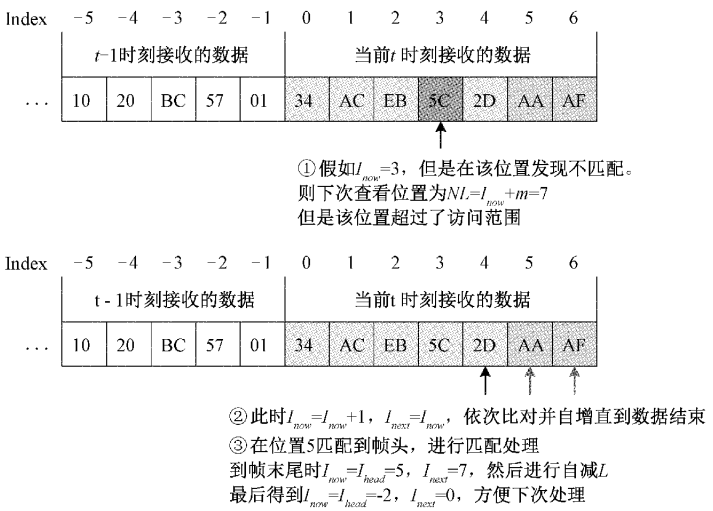

图 7 情况 1：在当前数据末尾执行进行依次比对

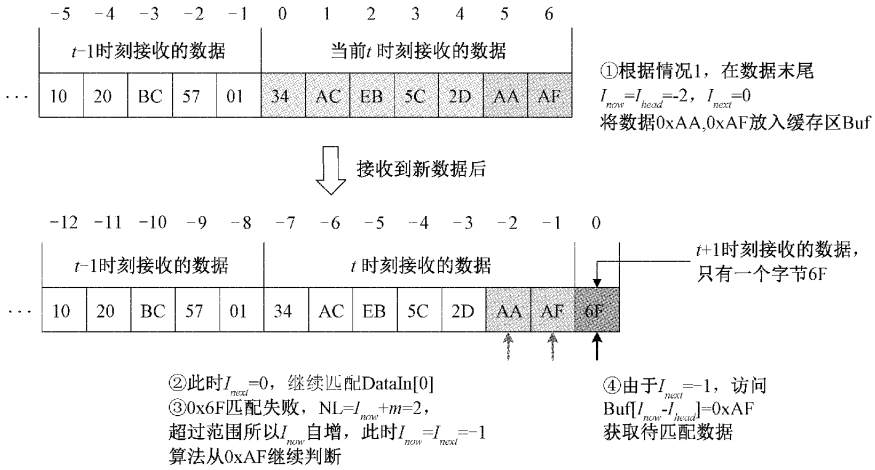

图 8　假帧头情况处理

帧内子域查找算法初始化

1：Input：匹配到的帧头位置 $StartIndex$，帧头长度 m
2：$I_{seg} \leftarrow StartIndex + m$
3：$L_{seg} \leftarrow 0$
4：设立帧缓存数组 ***SegBuf***

然后在 S3、S4 运行以下算法：(伪代码中 $Copy$ 为复制内存函数，用于数据间复制，输入参数为源数据，源数据起始复制偏移，目标数据，目标数据起始复制偏移，复制字节个数)。

帧内子域查找算法

1：Input：接收字节数组 ***DataIn***，接收数据长度 L，想要获取的子域在帧内偏移量 O_{get}，想要获取的数据段长度 L_{get}
2：Output：得到的帧内子域 ***DataFiledOut***
3：$O_{end} \leftarrow O_{get} + L_{get} - 1$
4：**if** $O_{get} \geqslant L_{seg}$ **then**
5：　**if** $I_{seg} + O_{end} \geqslant L$ **then**
6：　　$L_{copy} \leftarrow L - I_{seg} - L_{seg}$
7：　　$Copy$(***DataIn***，$I_{seg} + L_{seg}$，***SegBuf***，L_{seg}，L_{copy})
8：　　$L_{seg} \leftarrow L_{seg} + L_{copy}$
9：　　$I_{seg} = I_{seg} - L$
10：　　**return** null
11：　**else**
12：　　$L_{copy} \leftarrow O_{end} - L_{seg} + 1$
13：　　$Copy$(***DataIn***，$I_{seg} + L_{seg}$，***SegBuf***，L_{seg}，L_{copy})
14：　　$L_{seg} \leftarrow L_{seg} + L_{copy}$
15：　　**return** ***SegBuf***$[O_{get}, O_{get} + L_{get}]$
16：　　**end if**
17：**else**
18：　　**return** ***SegBuf***$[O_{get}, O_{get} + L_{get}]$
19：**end if**

其算法的主要思想可以用流程图表示，如图 9 所示，即根据数据访问范围来进行输出和缓存。

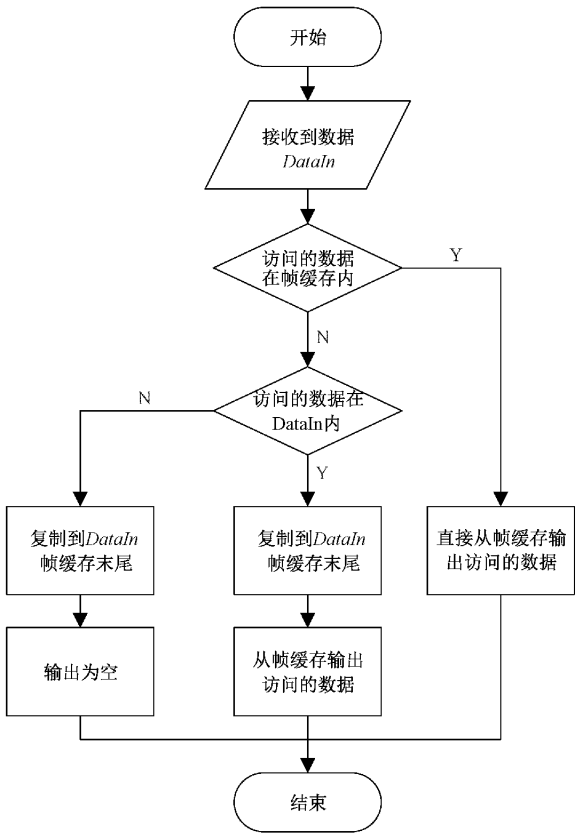

图 9　帧内子域查找算法流程

5　算法性能分析与测试

对于该算法而言，影响算法性能的主要因素有帧本身长度、帧头长度、单次接收的字节数组的大小等。为了测试该算法的整体性能，本文将采用两种数据输入方法：直接模拟数据输入，以及模拟数据经由局域以太网 TCP 输入。

5.1 直接模拟数据输入

为了模拟数据输入,本文首先设计了模拟帧生成器,生成器会生成大量数据帧,生成的帧的帧数据区大小由 4 字节帧长度域指定,一帧数据由帧头、帧长度域、帧数据区 3 部分构成。

其次,本文用本算法以及通常的直接的状态机跳转的方法(如图 3)分别设计了两种帧数据提取器。实验中,两种算法均在 Visual Studio 2022 平台下使用 c# 语言编写,并运行于 CPU 为 i7-10750H,2.60 GHz 的计算机上,实验将生成器产生的数据帧进行数组合并,并按照随机的字节数组大小进行重新分割,然后分别依次输送给两种帧数据提取器,实验统计从提取到第一帧到最后最后一帧提取成功的时间,单位为 ms。

本文首先以帧数据区长度为自变量,在帧头大小为 4 byte,发送 100 000 帧,接收字节数组的大小随机(范围为 1～1 000 byte)情况下,统计两种算法运行 50 次的平均处理时间,结果如图 10 所示。

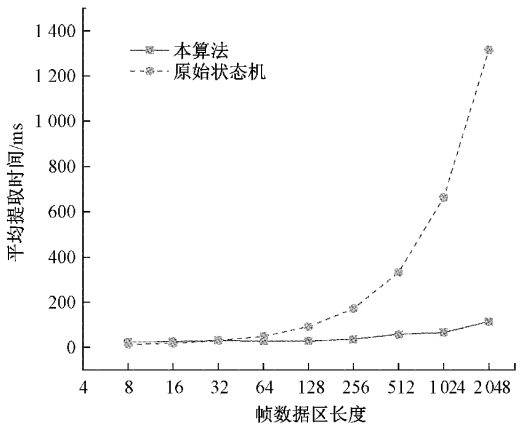

图 10 不同帧数据区长度下算法的平均耗时

可以从图 10 中看出,随着帧长度的增加,本算法和的平均耗时增长速度要小于直接使用状态机算法,这是因为对于帧内数据区的提取,整体复制可以交由计算机内部由更优化的方法进行提取,如嵌入式计算机可直接用直接存储器访问(direct memory access,DMA)直接完成内存到内存的后台复制,这种方法要比挨个进行复制要更快。同时,由于挨个复制在数据区较小时运行步数还是较小的,因此可以看出在帧数据区长度为 8、16 时使用原始的状态机跳转还是比较快的。

另外本文以接收字节数组的大小为自变量,在帧头大小为 4 byte,发送 100 000 帧,帧数据区大小随机(范围为 0～100 byte)情况下,统计两种算法的平均处理时间,结果如图 11 所示。

可以从图 11 中看出,随着接收数据大小的增加,算法的平均耗时逐渐下降并趋于稳定,这是因为对于两种算法,接收数组大小的增加会使得进入函数并处理数据边界

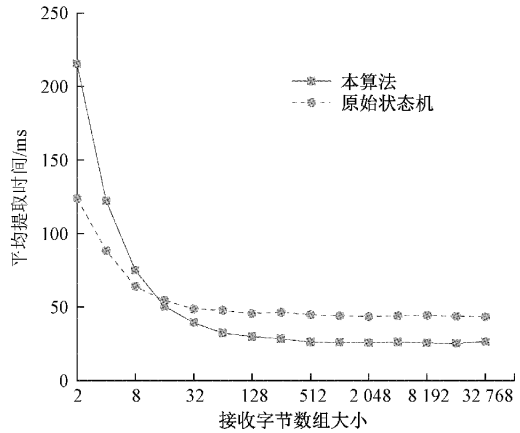

图 11 不同接收字节数组大小下的算法平均耗时

的耗时占比逐渐减小。在接收数据大小为 2、4、8 等较小值时,算法主要完成对边界的处理,这部分处理时间是相对固定的。经计算,在帧长度范围为 0～100 字节,接收字节数组大小大于 64 字节情况下,本算法的性能提升率趋于稳定,平均可提升 40.1%。

5.2 经由网络 TCP 输入

为了更全面地分析算法性能,模拟真实的总线数据接收环境,本文在相同的网络环境中与 Netty 网络框架下的长度域解码器(length field based frame decoder)进行比较测试。Netty 是一个高性能、异步的网络接口对象(network interface object,NIO)框架,该框架提供了基于固定长度解码器、特殊分隔符解码器、长度域解码 3 种解码方式用于处理网络数据包接收过程中的粘包、拆包问题[13-14]。

本文为了搭建实验环境,设立了 TCP 客户端和服务端,服务端绑定端口 8007。在本地局域网条件下,由 TCP 客户端在初始化时生成大量数据帧,帧格式和前文相同,然后在建立连接后发送数据帧给服务端[15]。在服务端,底层的接收采用相同的 Netty 框架下的代码,而在数据接收处理上本文将长度域解码器替换为本算法以进行测试。实验统计从提取到第一帧到最后一帧提取成功的时间,单位为 ms。

首先以帧数据区长度为自变量,在帧头大小为 4 byte,发送 100 000 帧,接收字节数组的大小随机(范围为 1～1 000 byte)情况下,统计 Netty 的长度域解码器和本算法的平均处理时间,结果如图 12 所示。

可以从图 12 中看出,随着帧长度的增加,两种算法的时间同步增加。经计算相对而言本算法的性能提升率为 28.55%。

由于接收字节数组的大小与实际网络环境和发送时间有关,因而本文测试客户端异步发送 10～100 k 帧数据情况下,服务端处理的耗时时间。在帧头大小为 4 byte,帧数据区大小随机(范围为 0～100 byte)情况下,统计两种算法的平均处理时间,结果如图 13 所示。

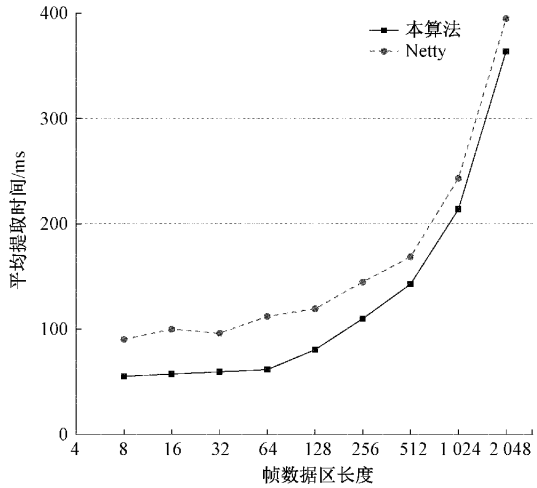

图 12　不同帧数据区长度下算法的平均耗时

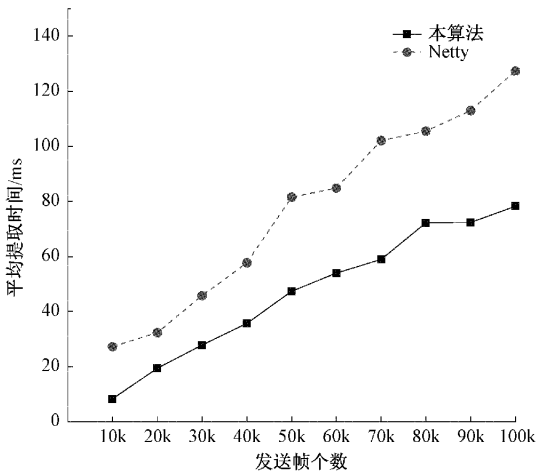

图 13　发送 10～100 k 帧数据情况下的算法平均耗时

可以从图 13 中看出，本算法整体快于 Netty 的帧长度域解析算法。另外，由于 Netty 的帧长度域解析器只能按照长度域进行帧的分隔。因而在实际测试中发现，若发送的第一帧不在接收数据的首位，会导致后续若干帧解析错误的现象。对此，若使用 Netty 算法进行动态长度解帧，必须确保首字节开始接收第一帧，否则需要特殊分隔符解码器、长度域解码两者配合使用，因而增加了算法的复杂性。

6　算法在高速模拟采集卡的应用

本文为了实际测试和应用算法，在某无人机测试项目中设计了 16 位，64 通道，100 kS/S 同步并行模拟量采集卡，该采集卡可通过千兆以太网向主控计算机每秒发送 800 000 条 UDP 数据帧，每条数据帧携带 8 个通道的模拟量数据。搭建系统结构如图 14 所示。

在该系统中，将函数信号发生器生成的 1 kHz，10Vpp 正弦波输入给模拟量采集板卡的第一个通道，并开启模拟量采集板卡的主动上报模式，将模拟量采样数据以 100 kS/s 的采样率发生给主控计算机。经测试，以太网平

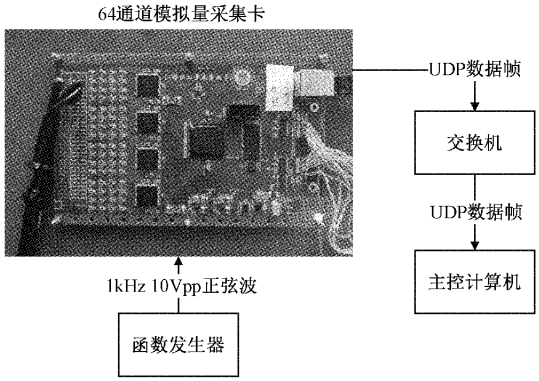

图 14　64 通道模拟量测试系统

均使用带宽可达 204 Mbps，使用 Wireshark 软件对网络数据进行监视，单条数据报文及如图 15 所示。

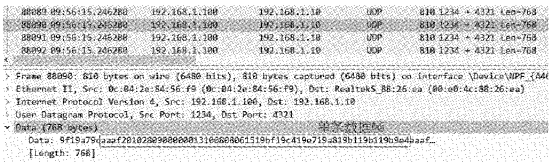

图 15　Wireshark 抓取的数据帧

在抓取可看到在单个 UDP 报文中有明显的粘包拆包现象，故可以使用该算法进行数据帧提取与分析。其数据帧格式如图 16 所示。

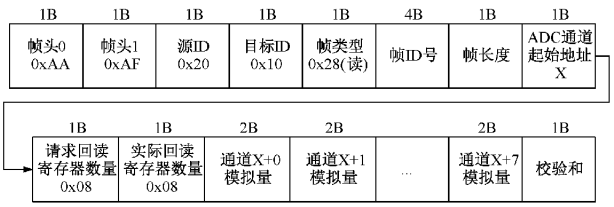

图 16　模拟量采集卡数据帧格式

为了解析该数据帧格式，可以在程序中实现帧处理类，使用改进的 Sunday 帧头匹配算法找寻帧头[0xAA，0xAF]，然后使用帧内子域查找算法找出帧长度信息，并根据帧长度信息提取全帧数据，最后使帧处理状态机跳转初始状态判定下一帧位置。若在以太网中接入了其他设备，也可以实例化新的帧处理类对数据流进行不同帧格式的帧提取。

本文基于 C # 的 WPF（windows presentation foundation）框架设计了上位机对数据使用该算法进行数据帧实时提取和存储，采集显示界面如图 17 所示。

其中对第一个通道采集模拟量的波形进行绘图，如图 18 所示，1 kHz 正弦波的单周期采样点数为 100 点，刚好对应 100 kS/s 采样率。

为了进一步评估信号波形，本文将采集到的波形数据导出，并使用进行频谱分析，如图 19 所示，同时根据参考文献[16]，计算得到其信号失真度为 0.061 2%，证明该算

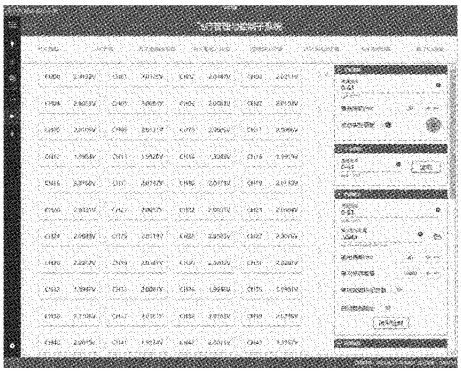

图 17 测试系统上位机界面

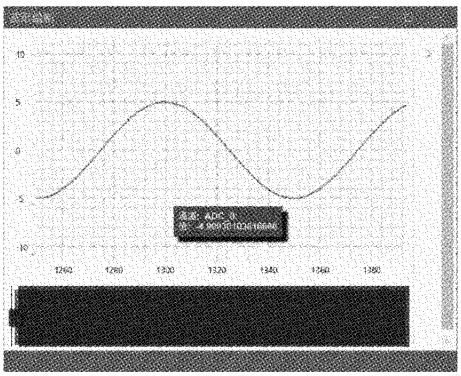

图 18 1 kHz 10Vpp 正弦波绘制

法实时提取的模拟量数据的波形完整,无明显失真,数据帧提取成功。

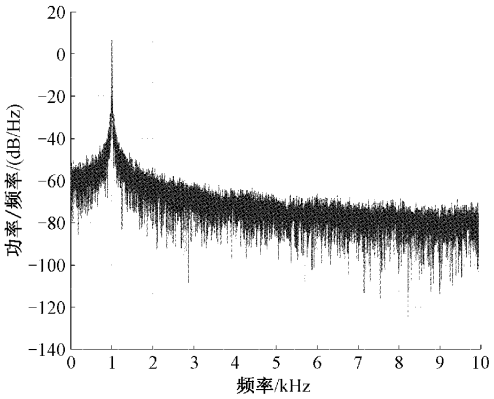

图 19 1 kHz 正弦波频谱分析结果

7 结 论

本文给出了常见的仪用总线网络的数据帧的协议处理方法,其中总线协议包括帧起始、帧地址、帧类型、数据校验、帧长度在内的 5 类总线帧处理的特性,并介绍了应用层通常用的状态机直接跳转的帧分析方法。

对此,本文设计了一套帧提取算法,算法包含帧提取状态机、改进的 Sunday 帧头匹配算法以及帧内子域查找三部分内容。然后本文与直接的状态机跳转方法、Netty 类似算法进行了基于 UDP 的数据帧处理实验结果比较,实验证明在帧长度较长时,本算法性能优于直接的状态机跳转方法,并整体优于 Netty 网络架构下帧长度域解码器。最后,本文为了实际测试和应用算法,使用 16 位,64 通道,100 kS/s 同步并行模拟量采集卡,在 PC 端使用该算法进行数据帧实时提取和存储,并对其采集的模拟量进行波形显示。本算法可用于各类仪用总线的应用层的数据分隔、帧头识别、帧数据提取工作。

参考文献

[1] 朱闻渊,尹家伟,蒋祺明. 新型航空电子系统总线互连技术发展综述[J]. 计算机工程,2011,37(S1):398-402.

[2] KJELLSSON J, VALLESTAD A E, STEIGMANN R, et al. Integration of a wireless I/O Interface for PROFIBUS and PROFINET for Factory Automation[J]. Ieee Transactions on Industrial Electronics, 2009, 56(10): 4279-4287.

[3] 王杨德. 面向比特流的协议帧头结构分析研究[D]. 上海:上海交通大学,2013.

[4] 赵恒永,路红武. 协议自适应的数据帧数据提取技术[J]. 计算机工程与设计,2006(4):701-703.

[5] 崔晓旻. 基于 Netty 的高可服务消息中间件的研究与实现[D]. 成都:电子科技大学,2014.

[6] 张艳军,王剑,叶晓平,等. 基于 Netty 框架的高性能 RPC 通信系统的设计与实现[J]. 工业控制计算机,2016,29(5):11-12,15.

[7] 苏锦. 基于 Netty 的高性能 RPC 服务器的研究与实现[D]. 南京:南京邮电大学,2018.

[8] GORDON R A, ABDEL H S, ALSAFRJALNI M H. A one-cycle FIFO buffer for memory management units in manycore systems[C]. 2019 IEEE Computer Society Annual Symposium on VLSI, 2019, 1: 265-270, DOI: 10.1109/ISVLSI.2019.00056.

[9] SUNG G M, TUNG L F, WANG H K, et al. USB transceiver with a serial interface engine and FIFO queue for efficient FPGA-to-FPGA communication[J]. Ieee Access, 2020, 8: 69788-69799, DOI: 10.1109/ACCESS.2020.2986510.

[10] HAKAK S I, KAMSIN A, SHIVAKUMARA P, et al. Exact string matching algorithms: Survey, issues, and future research directions[J]. Ieee Access, 2019, 7: 69614-69637, DOI: 10.1109/ACCESS.2019.2914071.

[11] RASOOL A, AHMED G F, BARSKAR R, et al. Efficient multiple pattern matching algorithm based on BMH: MP-BMH[J]. International Arab Journal of Information Technology, 2019, 16(6): 1121-1130.

[12] 朱宁洪. 字符串匹配算法 Sunday 的改进[J]. 西安科技大学学报，2016，36(1)：111-115.
[13] 王思源. 面向比特流的网络协议帧推理技术研究[D]. 武汉：武汉大学，2019.
[14] WANG D，DAI Y. Design of high-performance message middleware based on netty[C]. Journal of Physics：Conference Series，2022，1：012034，DOI：10.1088/1742-6596/2170/1/012034.
[15] 杨军，张和生，潘成，等. 一种交通信息采集传感器网络的 IP 互连方法[J]. 仪器仪表学报，2011，32(11)：2596-2601.
[16] 李翔，陈实. 时频结合的失真度测量方法研究[J]. 国外电子测量技术，2017，36(1)：27-30.

作者简介

范正吉，硕士研究生，主要研究方向为 FPGA 开发，三维重建，测控软件系统研制等。
E-mail：fzjfzj02@126.com

洪应平，博士，主要研究方向为究方向为极端环境下测试技术及仪器研究。
E-mail：hongyingping@nuc.edu.cn